SA Documentation

Tama Compiler Overhaul

Date: 2022-12-23 07:24:18+01:00
Semester: HS22

Students: Pascal Honegger
Marcel Joss
Leonard Schütz

Advisor: Philipp Kramer
Industry Client: Christian Marrocco

Table of Contents

Part I

Summary

Abstract

Triamec Motion AG manufactures ultra-precision servo drives that can be programmed using the C# programming language and the Tama framework. These servo drives run a virtual machine that executes a proprietary bytecode format, which is generated by the Tama compiler. The compiler previously relied on Microsoft's deprecated CCI library to read C# assembly files and transpile them to the Tama bytecode format. In this project, our goal was to replace the unmaintained CCI library with a modern, actively maintained alternative. After evaluating several options, we chose to replace the CCI library with ILSpy. ILSpy is a modern, actively maintained and well documented .NET decompiler with an active community.
In addition to ensuring the ability to support newer versions of .NET in the future, we also implemented a computed goto performance optimization in the servo's runtime, resulting in an overall performance increase of approximately 16%.
There is still potential for further performance gains, for example by implementing a template-based just-in-time (JIT) compiler in the servo's runtime.

Management Summary

Triamec Motion AG has developed ultra-precision servo drives that can be programmed using the C# programming language and the Tama framework. Recently the company sought to replace the unmaintained CCI library, which was used by the Tama compiler, with a modern, actively maintained alternative. Basing the Tama compiler on an unmaintained library poses several risks, such as not being able to support future versions of .NET and any bugs or other issues with the library not getting fixed.

After evaluating several options, ILSpy was chosen to replace the CCI library. This decision was based on several factors, including the client's prior good experience with ILSpy, the fact that many other projects depend on it, the strong community support for ILSpy, and the fact that it is maintained and actively developed. In comparison, every other alternative that was considered lacked at least one of these important aspects. In particular, this change ensures that the Tama compiler will be able to support newer versions of .NET in the future.

In addition to this update, several performance optimizations were implemented in the servo's runtime, which resulted in a performance increase of approximately 16%. This improvement allows customers to run more compute-intensive applications on their servo drives, or to run existing programs with a lower energy footprint, enhancing the company's competitive advantage in the market. There is still potential for further performance gains, such as by implementing a template-based JIT compiler in the servo's runtime.

Overall, the project has successfully updated the servo drives to support newer versions of .NET and has improved their performance.

Part II

Product Documentation

Chapter 1

Introduction

Triamec Motion AG is a Swiss company based in Baar, founded in 2001. The company is known for its outstanding knowledge and expertise in the field of mechatronics design and construction, as well as the regulation and control of highly dynamic systems. They offer consulting and development services for the implementation of ultra precision, high speed servo drives. The company's research and development team consists of engineers and physicists with decades of experience in the field of drive and control technology.

One of Triamec Motion AG's key technologies is the Tama ultra precision servo drive, which offers current and position control at up to 100kHz and real time programmable control at up to 10kHz cycles. The Tama framework and compiler provide a type-safe and standard way of programming the drive. The compiler translates the C# code from its intermediate language into the proprietary Tama bytecode format using Microsoft's CCI library.

However, since the CCI library is outdated and unmaintained, the company has recently taken interest in replacing the library with a modern, maintained alternative. The goal of this project is to evaluate several replacement candidates and to refactor the existing codebase to use the new library. Additionally, the project aims to implement optimizations that will improve the performance of the Tama virtual machine.

Chapter 2

Requirements

This section documents the main and optional requirements the team was given by the industry partner. These came in the form of extracts from their own support ticket platform, from which we then formulated our own requirements. Where applicable, the internal Tama issue tracking number is referenced.

2.1 Main Requirements

Requirements which should be prioritized and implemented in full.

Tama Compiler support for dotnet build (Triamec #1121)

The dotnet build pipeline should invoke the Tama Compiler directly as a process. There should be support for incremental builds. The Tama Compiler should be split into a .NET Standard 2.0 library and two separate executables.

Support for portable source symbol format (Triamec #442)

CCI does not provide support for the portable source symbol format. Switching to a modern alternative for CCI will allow customers to use portable PDBs by default. Support for portable PDBs is a requirement if the client ever wants to support a Tama library built on top of .NET Core, in order to support Linux.

Replacement of CCI library

The Tama Compiler relies on an unmaintained and deprecated library called CCI, which was developed by Microsoft. This library does not support modern .NET standards and newly found bugs won't be fixed. The library needs to be replaced with a modern alternative in order to support future .NET standards and to avoid running into bugs that the clients cannot fix themselves.

Compiler output should not change

The generated executables of the refactored compiler should not differ from the executables generated by the original compiler, other than superficial changes that do not have any impact on performance or behaviour.

All Unit-Test should pass

The unit-tests provided by the client should all pass successfully (if they passed before).

2.2 Optional Requirements

Requirements that can be prioritized should the time and effort be justifiable.

Virtual Machine Optimizations

To improve the performance of the Virtual Machine executing the Tama programs, several performance optimizations can be implemented. The main optimization that should be focused on, is the computed goto dispatch mechanism.

Method Inlining

Implement the method inlining compiler optimization. Method inlining refers to a technique by which the contents of a function body are copied to the call-site, removing the function call entirely. It should be determined whether it makes sense to place the requirement that the resulting executable should not be bigger than with the optimization turned off.

Constant Register Array Optimization (Triamec #472)

Replace occurrences of register element store and load operations with constant indexing register store and load counterpart operations.

Support instance base classes with fields (Triamec #667)

Support inheriting from base classes that define their own fields. Currently, the fields of inherited classes overlap the base class fields, resulting in unexpected and incorrect program behaviour.

Support static constructors in helper classes (Triamec #359)

Support the static constructors C# language feature.

Support array initializers (Triamec #1179)

Support the usage of array initializer literals. Will require the ability to store data blobs in the VM. Needs *ldtoken* support in the compiler and VM. Needs code for the *System.Runtime.CompilerServices.RuntimeHelpers.InitializeArray* function. Needs investigation for corner cases that cannot be supported.

Show meaningful errors when attempting to use 64-bit integer opcodes (Triamec #887)

Some integer operations in C# code might result in the generation of 64-bit integer opcodes. Some of these opcodes aren't supported by the Tama compiler and currently result in a cryptic error message. Catch these errors early on and produce a useful error message that helps in fixing the issue.

Chapter 3

Quality Measures

This section documents the measures the team has taken to assure proper quality of the final product.

3.1 Automated Test Suite

The Tama compiler already has a suite of integration tests which verify the output of the Tama VM when a program is executed. In addition they also verify the various compiler errors such as unsupported C# features as well as runtime errors that can be thrown by the VM.

3.2 Continuous Integration

Every commit pushed to the project repository is checked against our CI pipeline. Merge requests can only be merged into the main branch if all checks are successful. The following checks are part of the CI-pipeline:

- Building this LaTeXdocument
- Building the compiler
- Running the test suite

Since Tama currently only runs on Windows, the standard docker runners could not be used. Instead, a GitLab runner had to be set up manually on a Windows VM along with the necessary build tools and unit test runner.

3.3 Code Review

Using merge requests ensures that code is frequently reviewed by other project members, improving code quality.

3.4 Semi-Manual Output Testing

The integration tests only verify the behavior of the VM, but our goal is to produce the exact same output byte by byte as the previous version of the compiler. The output of the compiler consists of .tama and .asm files. The .tama files are the actual binaries which are executed by the VM, while the .asm files provide plain text output of the generated op codes. To verify that the generated op codes are identical the following process was applied:

1. The files generated by the previous version of the compiler are checked into a new Git repository
2. The files are overwritten by versions generated by the new version of the compiler and committed again
3. The command `git show *.tama` is run. If it has an empty output, it means that the compiler generated identical .tama files
4. If a .tama file differs, check the corresponding .asm file to find out what changed

This process was applied to all Tama executable files generated by the integration tests and four different projects provided by Triamec and repeated for the initial CCI abstraction layer and the ILSpy version.

Chapter 4

Evaluation

This section documents the evaluation phase. It consisted mainly of figuring out which library to replace the CCI library with.

4.1 CCI Alternative

The existing implementation of the Tama Compiler heavily relies on the Microsoft Common Compiler Infrastructure (CCI). Unfortunately, this CCI framework is officially deprecated and no longer maintained[1]. This technical debt is a long term risk for the Tama Compiler, so multiple ways to resolve this issue were analyzed.

Maintain CCI fork

One could create a fork of the existing CCI repository. This would require a significant time investment due to the complex implementation of CCI itself. This can be illustrated by a recent comment from a CCI contributor:

> The Microsoft CCI backend needs to be periodically updated to reflect the latest language changes which is often a time-consuming task requiring significant expertise.[2]

In addition to the sheer complexity, the GitHub repository also contains roughly 200k lines of code, more than the Tama project itself. Based on these factors, it was determined that maintaining and patching a fork is unsustainable in the long term.

[1] Microsoft, Microsoft/cci, 2018-07. [Online]. Available: https://github.com/microsoft/cci (visited on 2022-11-21).

[2] T. Kapin, Roslyn-based genapi backend · issue #10291 · dotnet/arcade, 2022-08. [Online]. Available: https://github.com/dotnet/arcade/issues/10291 (visited on 2022-11-21).

Mono Cecil

Mono Cecil[3] doesn't appear to be in active development, with its last official release about a year ago. The last update to the developer blog was uploaded about three years ago. The documentation also seemed a bit lacking, with only five code-snippets to be found in total. We concluded that switching to Mono Cecil wouldn't be sustainable, as it would be out-of-date again within a couple years.

System.Reflection.Metadata

Microsoft provides a built-in .NET toolset to access low-level assembly metadata information called System.Reflection.Metadata[4]. This library can be used to extract type-information, opcode-listings and other assembly-file metadata information from a compiled assembly-file. The functionality provided is pretty abstract, meaning we would have to implement a lot of the required logic ourselves. Since other libraries, such as ILSpy, already implement this logic by using System.Reflection.Metadata under the hood, we concluded that working with it directly doesn't make sense for our use-case and would only provide unnecessary work.

Use Roslyn Compiler

A conceptually different approach could be implemented using the official .NET Compiler Platform, aka Roslyn[5]. The Roslyn Compiler provides an API to traverse the Syntax Tree emitted by the Parser. Since the abstraction-level of the library is higher than that of the currently used CCI library, we would have to re-implement a lot of logic by ourselves. We concluded that switching to the Roslyn Compiler would cause too much work and would needlessly increase project complexity.

[3] J. Evain, Jbevain/cecil. [Online]. Available: https://github.com/jbevain/cecil (visited on 2022-11-21).

[4] Microsoft, System.reflection.metadata. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/api/system.reflection.metadata (visited on 2022-12-08).

[5] . Platform, Dotnet/roslyn. [Online]. Available: https://github.com/dotnet/roslyn (visited on 2022-11-21).

ILSpy

ILSpy[6] is a library that is being actively developed as open-source on GitHub. The library uses System.Reflection.Metadata under the hood and can be used to decompile projects to C# source code. There also exists an API to iterate over each IL-Opcode, which is exactly what we would need to re-implement the functionality of the Tama Compiler. However, some C# language features aren't implemented yet, as can be seen via the projects language support checklist[7]. This library seems to be the most promising candidate.

Decision

In discussion with our business partner, it was decided to remove the dependency upon CCI. The technical risk of keeping CCI as a dependency was deemed unmaintainable. This drive to remove CCI was further fueled by the fact that many other open source libraries on GitHub are actively migrating away from CCI. In lieu of this decision, it was obvious to use ILSpy due to a number of factors:

- Client has prior good experience with ILSpy
- Plenty of other projects depending on it, great community
- Maintained and actively developed

Every other project lacks at least one of these aspects, which are all crucial for the longevity of the Tama compiler.

[6] D. G. Siegfried Pammer, Icsharpcode/ilspy. [Online]. Available: https://github.com/icsharpcode/ILSpy (visited on 2022-12-08).

[7] D. Grunwald, Icsharpcode/ilspy: Issue #829, 2017-08. [Online]. Available: https://github.com/icsharpcode/ILSpy/issues/829 (visited on 2022-12-08).

Chapter 5

Final Result

This section documents the final state of all the changes the team made. It documents the abstraction layer, the replacement of the CCI library and the optimizations done in the Tama virtual machine.

5.1 Abstraction Layer

Building an abstraction layer was chosen for several reasons. Creating wrapper objects for library objects decouples the business logic and code generation from the parsing library. It also provided a great opportunity for the team to become familiar with the codebase. This separation of responsibilities helps to reduce technical risk in case the parsing library does not work as expected.
By starting with the CCI implementation and running tests to ensure that the abstraction itself works, the library can then easily be replaced with ILSpy with minimal changes to the code outside of the parsing. This allows for switching between different parsing libraries without having to make major changes to the rest of the code, which is especially useful in case any issues were encountered with ILSpy.
Overall, building an abstraction layer provides a number of benefits, including better code organization, reduced technical risk and the ability to easily switch between different parsing libraries. This will ultimately save time and effort in the long run and make the project more maintainable and scalable. If one day the ILSpy library is no longer maintained, the abstraction layer would make it a lot easier to switch to a different library.

Class Model

A hierarchy of interface inspired by CCI was created but only using those interfaces and members which were actually used in the codebase. At the base of the hierarchy is the `IMetadataAcceptor` interface which defines the `Accept` method for use with visitors. Objects such as types and methods each get a reference and a definition interface. Some objects such as parameters and local variables only exist as definitions while others like

array and pointer types only exist as references. The implementations of those interfaces typically wrap an object from the library (CCI or ILSpy) and forward their properties while wrapping them in an abstraction layer object. The following diagram illustrates the hierarchy with the example of a method definition.

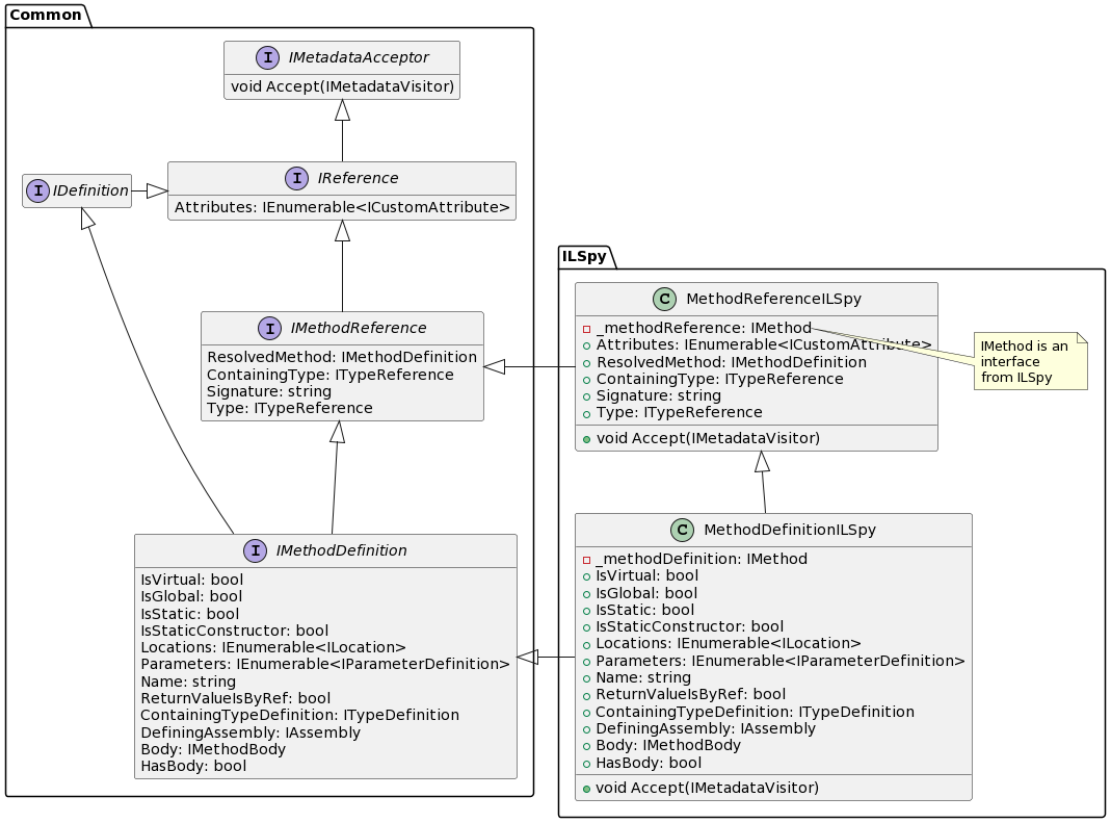

Figure 5.1: Excerpt of the abstraction layer class hierarchy

Visitor Pattern

The existing implementation relied heavily on CCI to parse the input of the Tama compiler. CCI itself provides a visitor for this task, making it easy to extend for Tama's purposes. To keep the impact on existing code minimal, a generalized visitor class was implemented based on the new abstraction class model. The new visitor implements the same logic as the one present in CCI, but requires much less code. The end result is a library-independent and more comprehensible abstraction layer, which helps with the understanding of its implementation.

Null Object Pattern

The CCI library applies the null object pattern[1] thoroughly. Instead of simply returning a null value to indicate that an object is not present, the null object pattern returns a 'dummy' object with default values. This can make it more difficult to reason about the code and to understand what is happening at a given point in the program. Additionally, implementing the null object pattern would require a significant amount of additional code, which can make the codebase more complex and difficult to work with. A possible way of implementing dummy objects in our model can be seen in 5.2

```
/** Abstraction layer interface */
interface IAsssemby {
    string Path { get; }
}

/** Abstraction layer dummy implementation */
class DummyAsssemby : IAsssemby {
    string Path => "Dummy Path";
}

/** Library specific implementation */
class AsssemblyILSpy : IAsssemby {
    string Path => /* ... */;
}
```

Figure 5.2: Example of possible custom dummy implementation.

In the context of the Tama compiler, using null values is a more straightforward and less error-prone way of representing the absence of an object. Therefore, the existing implementation was refactored to use null values instead of 'dummy' objects. This change required significant modifications to the codebase, but we believe that it was worth it in the long run. The resulting code is easier to understand and maintain, and it eliminates the need for custom 'dummy' objects. An example of how the code was changed can be seen by comparing figure 5.3 and figure 5.4.

[1] W. contributors, Null object pattern, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Null_object_pattern&oldid=1091628852 (visited on 2022-12-12).

```
private IAssemblyReference _assemblyReference =
    Dummy.AssemblyReference;
void GetAssemblyReference() {
    if (_assemblyReference == Dummy.AssemblyReference) {
        _assemblyReference = new AssemblyReference(/* ... */);
    }
    return _assemblyReference;
}
```

Figure 5.3: Example of code using CCI dummies.

```
private IAssemblyReference _assemblyReference;
void GetAssemblyReference() {
    if (_assemblyReference is null) {
        _assemblyReference = new AssemblyReference(/* ... */);
    }
    return _assemblyReference;
}
```

Figure 5.4: Example of code using null.

5.2 CCI Replacement - ILSpy

As decided in chapter 4.1, the CCI dependency was to be replaced with ILSpy. Thanks to the previously implemented abstraction layer, there was a clearly defined API contract to implement for this task. We organized the development in such a way that we proceeded in a structured way, implementing one use case at a time instead of doing everything at once. This process ensured that the scope of the new ILSpy-based parser could be flexibly updated if unexpected issues arose.

Restructuring of provided data

Some object models within ILSpy made no sense for our use-case, so we decided to map them to our existing abstraction instead of altering the abstraction itself. For example, a custom type for array type references existed in CCI and is implemented in our abstraction layer. Contrary to those implementations, ILSpy provided the member type instead of the array type for certain operation codes like 'NewArr'.

Iterating over Opcodes

ILSpy provides an `ILVisitor` class, which can be extended to iterate over the op codes. However, we found that it's data model was already too abstract. For example, all binary numeric operations such as addition, subtraction, multiplication, and division become a `BinaryNumericInstruction` object. Figure 5.5 shows an example of the extra work required to get from ILSpy's objects to the original op code.

```
protected override void
    VisitBinaryNumericInstruction(BinaryNumericInstruction inst) {
     switch (inst.Operator) {
          case ICSharpCode.Decompiler.IL.BinaryNumericOperator.Add:
             {
              ILOpCode op;
              if (inst.CheckForOverflow) {
                   op = inst.Sign.IsSigned() ? ILOpCode.Add_ovf :
                      ILOpCode.Add_ovf_un;
              } else {
                   op = ILOpCode.Add;
              }

              AddOp(inst, op);
              break;
          }
          // Rest omitted
}
```

Figure 5.5: Example of ILSpy `ILVisitor`

This procedure is unnecessarily complicated and prone to errors. There are also cases in which mapping back to the original op code is virtually impossible without being extremely coupled to the ILSpy implementation. For these reasons `ILVisitor` turned out not to be useful for our use case and the team switched to working directly with the raw byte stream. ILSpy still proved itself a great help thanks to its utilities for decoding op codes and their arguments from the raw binary data.

Debug Information

Debug information is needed for the names of local variables and for finding the source code location of an intermediate language instruction. The ILSpy NuGet package includes an interface for debug information, but no implementations of it are included in the package itself. There is another NuGet package called ILSpyX which does provide this functionality, but it's only available for .NET 6 and is still in a preview version. We opened an issue on ILSpy's repository[2] to find a solution and a maintainer recommended simply copying the needed classes from the GUI version of ILSpy. This approach requires a dependency on Mono.Cecil as well as allowing unsafe blocks in the project. For these reasons, we decided not to merge this into the project's main branch, but to leave the decision to Triamec.

[2] P. Honegger, Icsharpcode/ilspy: Issue #2865, 2022-12. [Online]. Available: https://github.com/icsharpcode/ILSpy/issues/2865 (visited on 2022-12-21).

5.3 Computed Goto

The basic approach to implementing a bytecode-interpreter is to have a central switch-statement wrapped inside a while-loop. The case blocks contain the implementation of each opcode. After an opcode-handler has finished executing, control-flow continues back at the beginning of the switch-statement, which decides which opcode-handler to pass control to next. Figure 5.6 shows an example of a bytecode interpreter written in C, following the while-switch pattern.

```
int interpreter_switch(int* code, int pc, int accumulator) {
    while (true) switch (code[pc++]) {
        case OP_HALT: {
            return accumulator;
        }
        case OP_ADD_ARG: {
            accumulator += code[pc++];
            continue;
        }
        case OP_MUL_ARG: {
            accumulator *= code[pc++];
            continue;
        }
    }
}
```

Figure 5.6: Example of a bytecode interpreter following the while-switch pattern.

Hardware

Modern microprocessors perform a technique called pipelining, which refers to the CPU preemptively loading sections of the code it predicts will be executing next. When combined with another component, called the branch predictor[3], this technique can result in considerable performance gains. This is due to the fact that the CPU can now load contiguous chunks of code memory at once, as opposed to fetching each opcode individually.

Shortcomings

In order to understand how the interaction between the branch predictor and the switch statement wrapped in a while loop leads to performance inefficiencies, one must first understand how the compiler generates code for it. In unoptimized builds, or for small switch statements with only a few conditions, the compiler emits a series of if statements, each checking for a single condition and then jumping to the relevant case block. Some optimizing compilers might instead emit a jump table, containing the addresses of the

[3]W. contributors, Branch predictor, 2022-11. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Branch_predictor&oldid=1119722013 (visited on 2022-11-21).

case blocks, indexed by the switch's condition. The branch-predictor builds an internal cache which stores the frequencies at which each specific jump is taken, in the hopes of speeding up the transfer to the relevant case block. However, because the same checks are performed before each opcode, the branch predictor's cache is quickly saturated and previous entries are overwritten with newer ones. The ability to determine which opcode tends to follow another is lost or at least greatly diminished in efficiency. (Note: This is a simplified explanation of how the branch-predictor operates, however it is sufficient to understand the problem).

Another minor inefficiency is the need to perform a bounds check during the switch statement evaluation. Even if the compiler decides to emit the switch statement in the form of a jump table, the program must still check that the condition value doesn't exceed the size of the jump table. Unimplemented or invalid opcodes might get emitted into the instruction stream, either through a compiler bug or random memory corruption, which is an event that needs to be handled by the interpreter gracefully.

Computed Goto

Computed gotos aim to solve this problem by providing each opcode-handler with it's own dispatch mechanism. It is a combination of two lesser-known features in the C language family. The first is taking the address of a label, and storing it in a variable. The second is calling **goto** with a variable expression, instead of a label defined at compile-time. This allows the creation of a dispatch-table, containing the starting addresses of each opcode handler. At the end of each opcode-handler, the table is indexed by the next opcode and control-flow is passed directly to that address. This circumvents the central switch-statement and allows the hardware branch-predictor to set up a cache for each individual opcode handler. Because the size and contents of the dispatch-table can be precisely controlled, the need for a bounds-check falls away. By filling the unused portions of the table with the addresses of the default-handler, any possible value that the opcode could take, even invalid ones, are covered by the dispatch-table. Because all possible opcode values are now handled, the code for the bounds-check can be safely removed, reducing overall instruction count and improving throughput. Combining these factors, it is common for this optimization to result in a 15–20% improvement in bytecode throughput[4]. Figure 5.7 shows an example of a bytecode interpreter written in C, following the computed goto pattern.

[4]E. Bendersky, "Computed goto for efficient dispatch tables," 2012-07. [Online]. Available: https://eli.thegreenplace.net/2012/07/12/computed-goto-for-efficient-dispatch-tables (visited on 2022-11-21).

```
int interpreter_computed_goto(int* code, int pc, int accumulator) {
    int opcode = 0;

    static void* dispatch_table[] = {
        &&opcode_handler_halt,
        &&opcode_handler_add_arg,
        &&opcode_handler_mul_arg
    };

#define DISPATCH_NEXT()             \
    opcode = code[pc++];            \
    goto* dispatch_table[opcode]

    DISPATCH_NEXT();

opcode_handler_halt:
    return accumulator;

opcode_handler_add_arg:
    accumulator += code[pc++];
    DISPATCH_NEXT();

opcode_handler_mul_arg:
    accumulator *= code[pc++];
    DISPATCH_NEXT();
}
```

Figure 5.7: Example of a bytecode interpreter following the computed goto pattern.

Implementation

Implementing the computed goto optimization in the Tama VM turned out to be more complex than initially assumed. Since the VM is compiled with MSVC, we couldn't use the computed goto C language-extension. The next approach was to implement the required dispatch-mechanism manually, using inline assembly snippets. This technique worked out great for debug builds, which were running in 32-bit mode, but failed to compile for 64-bit target architectures. This was due to MSVC's fundamental lack of inline-assembly support for 64-bit platforms. Because the VM runs on 64-bit hardware, this approach turned out to be a dead-end.
As a last resort, the dispatch-mechanism was implemented using regular C language control structures, namely a switch-statement at the end of each opcode handler, mimicking the mechanism that the computed goto extension would provide. While this implementation worked on 64-bit targets, the produced binary showed an increase in file size of over 500%. When compiled in release-mode, the compiler would even crash due to heap memory exhaustion. The cause of this massive increase in file size was the fact that the compiler would emit hundreds of identical copies of the dispatch table into the final executable. During a discussion with the client, this increase in file size was deemed to be acceptable and a performance test was performed.

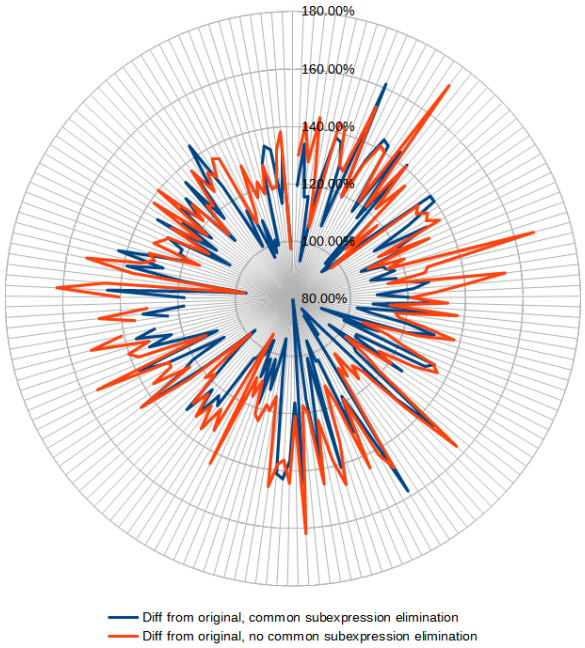

Figure 5.8: Average change in opcode execution time, using naïve manual MSVC computed goto implementation, figure provided by client

As shown in figure 5.8, the modifications resulted in a decrease in performance for most opcodes. Depending on whether the common-subexpression-elimination optimization

was activated or not, the execution time for each opcode grew by an average of 18–28%. Computing time for more complex tests also grew by about 10%.

Switch to GCC

The client initially wanted to scratch these modifications, as the performance test clearly showed a performance-decrease. After some discussions, our team was made aware of the fact that the production build of the VM, running on actual hardware, is being compiled using GCC instead of MSVC. GCC has native support for the computed goto C language-extensions, which is exactly what's needed here. This allowed us to maintain two different versions of the VM, one compiled with MSVC and the other with GCC. The MSVC version would be used whenever the VM was executed in a simulated environment, as performance wasn't of high priority there. The GCC version was to be used on real-world hardware, since performance mattered the most there. Since the client is maintaining two different versions of the VM source code already, we did not need to implement the optimization in a compiler-independent way.

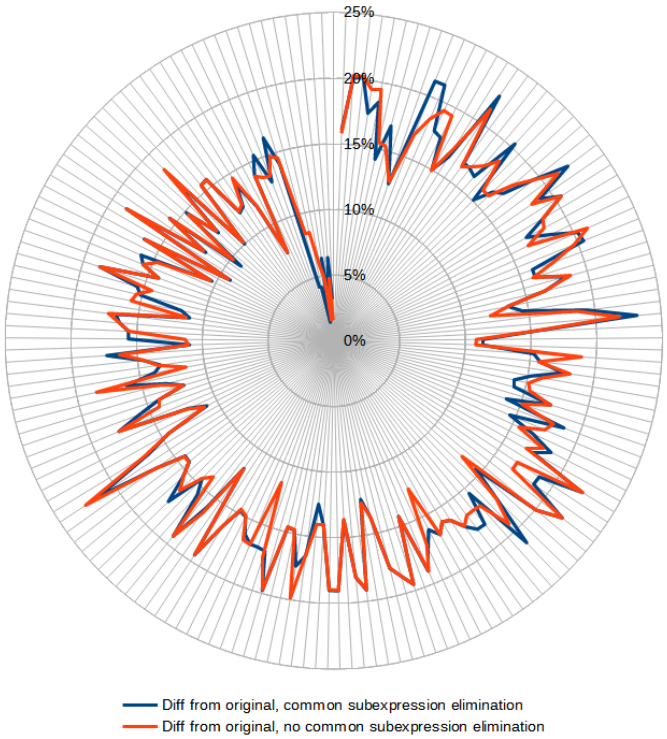

Figure 5.9: Average reduction in opcode execution time, using built in GCC computed goto extension, figure provided by client

As shown in figure 5.9, this change resulted in the expected performance boost. The execution time for individual opcodes was decreased in all cases, on average by 16% with

a standard deviation of 4%.

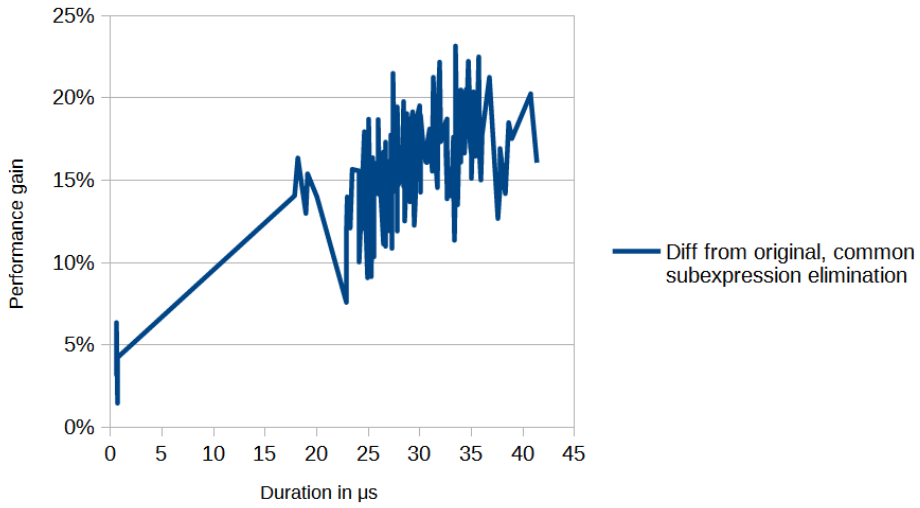

Figure 5.10: Average performance gain of tama programs, depending on execution-time, figure provided by client

Figure 5.10 shows the performance gain for more realistic and complex test-programs. Programs that ran for longer, had a greater increase in performance than those that ran for shorter lengths of time. On average, the performance of these programs was increased by about 16%.

Conclusions

The computed goto optimization has improved runtime-performance in a clear, significant and reproducible manner. The client was pleased with these results and informed us that these modifications will be included in the next release of their production firmware. This improvement in efficiency will benefit all customers by allowing them to run more computationally intensive applications on their machine, or run existing programs with a lower energy-footprint, increasing the company's competitive edge in the market.

Graphs and Test Results

Graphs showing results of performance-analysis were provided by the client. All performance tests were conducted on real, production hardware.

5.4 Conclusion

Overall, the main requirements were implemented in the expected scope. Our code quality efforts, including automated unit tests and semi-manual comparison of compiler outputs, have confirmed that the refactored implementation is correct and produces the same Tama executables as the original version.

Abstraction Layer An abstraction layer was built inside the Tama Compiler, leading to better separation of concerns and easier maintainability in the future. If the ILSpy library becomes unmaintained as well, it should be an easy task to replace it with a modern version in the future.

Replacing CCI with ILSpy The dependency on Microsoft's CCI library was eliminated completely. The unmaintained library was replaced with ILSpy, which covers the vast majority of the previous's library features and use cases. There was however a slight feature regression in regard to the displaying of error information, because the way ILSpy handles debug metadata differs from CCI. Reintegrating this feature shouldn't pose too big of a technical challenge.

Computed Goto Optimization The switch-elimination (Computed Goto optimization) was implemented successfully in the Tama Virtual Machine. On average production workloads, the optimization has improved performance (opcode throughput) by 16%. However, the optimization is only implemented for the production build of the Tama Virtual Machine (compiled by GCC). It is not available for the debug build that is used for the execution of the runtime's unit tests. This is due to missing compiler support from MSVC.

SDK Style Project Support for SDK style projects has been implemented. The build now relies on VS2022 for much of its functionality. We increased the versions of the Windows 10 SDK and the MSVC C++ compiler. This was done solely for our convenience and should the client wish to downgrade to an older version again, they would still be able to do so.

Portable PDB Support Support for portable PDB files has been implemented by switching to ILSpy. Since ILSpy supports this functionality natively, this required little new code to be written. This functionality is verified by automated tests.

Tama Compiler support for dotnet build Initially, support for the dotnet build system was impossible due to the usage of the CCI library. Even after replacing the CCI library with ILSpy, support for dotnet build is still not fully implemented. The team has however eliminated most major hurdles within the codebase that would've prevented a switch to target .NET 7 and .NET 4.6.2. Some remaining problems with support for dotnet build are:

- Limited .NET Standard support within the *Triamec.Tam.Core* NuGet package.
- DLL version mismatch within ILSpy when targeting .NET 4.6.2 and .NET 7
- Automated unit test projects should be migrated to .NET 7

Some of these outstanding problems were addressed but not verified or tested yet. Further investigation into these topics is left to Triamec.

Glossary

CCI Microsoft Common Compiler Infrastructure library.

CLI The Common Language Infrastructure is a technical standard developed by Microsoft, describing the runtime environment used to execute C# code.

GCC Optimizing compiler developed by the Free Software Foundation.

IL The C# Intermediate Language is a platform-independent instruction set that can be executed on any architecture that supports the CLI.

ILSpy ILSpy is a .NET Decompiler with support for PDB generation.

JIT A Just In Time compiler generates architecture specific code on demand.

MSVC Microsoft Visual C++ compiler toolchain.

PDB Program database files used by the Visual Studio debugger to store information about variables, functions, symbols and types in a program.

VM A Virtual Machine is a simulated machine that runs within a program.

List of Figures